5 Key exchanges

This chapter covers

- What key exchanges are and how they can be useful
- The Diffie-Hellman and Elliptic Curve Diffie-Hellman key exchanges
- Security considerations when using key exchanges

We are now entering the realm of *asymmetric cryptography* (also called *public key cryptography*) with our first asymmetric cryptographic primitive: the *key exchange*. A key exchange is, as the name hints, an exchange of keys. For example, Alice sends a key to Bob, and Bob sends a key to Alice. This allows the two peers to agree on a shared secret, which can then be used to encrypt communications with an authenticated encryption algorithm.

> **WARNING** As I hinted in the introduction of this book, there is much more math involved in asymmetric cryptography; therefore, the next chapters are going to be a tad more difficult for some readers. Don't get discouraged! What you will learn in this chapter will be helpful to understand many other primitives based on the same fundamentals.

NOTE For this chapter, you'll need to have read chapter 3 on message authentication codes and chapter 4 on authenticated encryption.

5.1 What are key exchanges?

Let's start by looking at a scenario where both Alice and Bob want to communicate privately but have never talked to each other before. This will motivate what key exchanges can unlock in the simplest of situations.

To encrypt communications, Alice can use the authenticated encryption primitive you learned about in chapter 4. For this, Bob needs to know the same symmetric key so Alice can generate one and send it over to Bob. After that, they can simply use the key to encrypt their communications. But what if an adversary is passively snooping in on their conversation? Now the adversary has the symmetric key and can decrypt all encrypted content that Alice and Bob are sending to each other! This is where using a key exchange can be interesting for Alice and Bob (and for ourselves in the future). By using a key exchange, they can obtain a symmetric key that a passive observer won't be able to reproduce.

A *key exchange* starts with both Alice and Bob generating some keys. To do this, they both use a key generation algorithm, which generates a *key pair*: a private key (or secret key) and a public key. Alice and Bob then send their respective public keys to each other. *Public* here means that adversaries can observe those without consequences. Alice then uses Bob's public key with her own private key to compute the shared secret. Bob can, similarly, use his private key with Alice's public key to obtain the same shared secret. I illustrate this in figure 5.1.

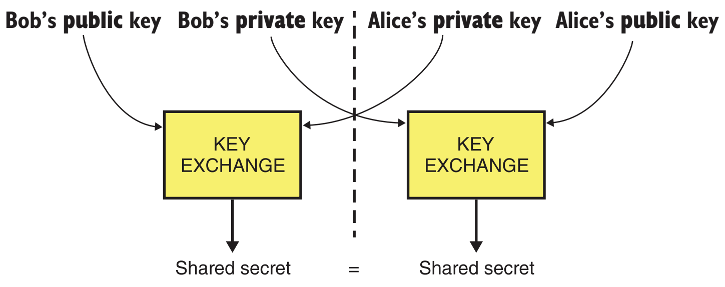


Figure 5.1 A key exchange provides the following interface: it takes your peer's public key and your private key to produce a shared secret. Your peer can obtain the same shared secret by using your public key and their private key.

Knowing how a key exchange works from a high level, we can now go back to our initial scenario and see how this helps. By starting their communication with a key exchange, Alice and Bob produce a shared secret to use as a key to an authenticated encryption primitive. Because any man-in-the-middle (MITM) adversaries observing the exchange cannot derive the same shared secret, they won't be able to decrypt communications. I illustrate this in figure 5.2.

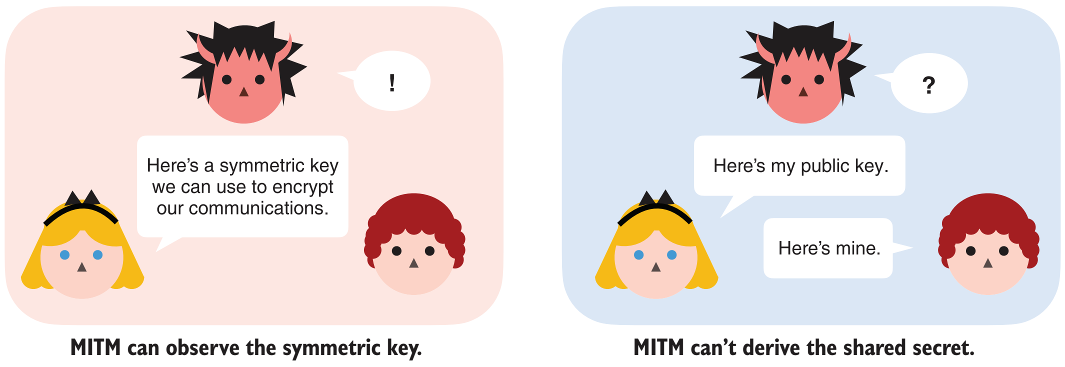

!
Here's a symmetric key we can use to encrypt our communications.
MITM can observe the symmetric key.
?
Here's my public key.
Here's mine.
MITM can't derive the shared secret.

Figure 5.2 A key exchange between two participants allows them to agree on a secret key, while a man-in-the-middle (MITM) adversary can't derive the same secret key from passively observing the key exchange.

Note that the MITM here is passive; an *active* MITM would have no problem intercepting the key exchange and impersonating both sides. In this attack, Alice and Bob would effectively perform a key exchange with the MITM, both thinking that they agreed on a key with each other. The reason this is possible is that none of our characters have a way to verify who the public key they receive really belongs to. The key exchange is *unauthenticated*! I illustrate the attack in figure 5.3.

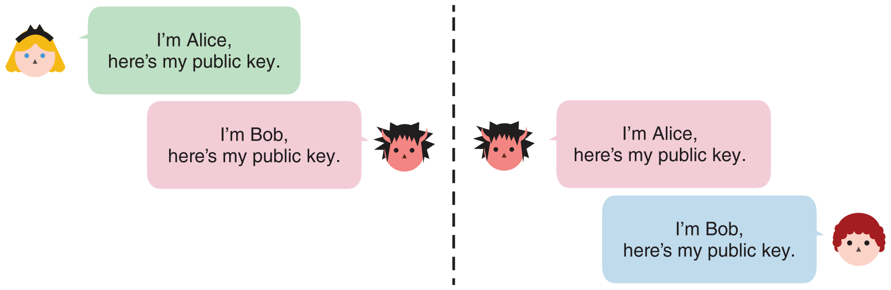

I'm Alice, here's my public key.
I'm Bob, here's my public key.
I'm Alice, here's my public key.
I'm Bob, here's my public key.

Figure 5.3 An unauthenticated key exchange is vulnerable to an active MITM attacker. Indeed, the attacker can simply impersonate both sides of the connection and perform two separate key exchanges.

Let's take a look at a different scenario to motivate *authenticated key exchanges*. Imagine that you want to run a service that gives you the time of day. Yet, you do not want this information to be modified by a MITM adversary. Your best bet is to authenticate your responses using the message authentication codes (MACs) you learned about in chapter 3. As MACs require a key, you could simply generate one and share it manually with all of your users. But then, any user is now in possession of the MAC key you're using with the others and might some day make use of it to perform the previously discussed MITM attack on someone else. You could set up a different key per user, but this is not ideal as well. For every new user that wants to connect to your service, you will need to manually provision both your service and the user with a new MAC key. It would be so much better if you didn't have anything to do on the server side, wouldn't it?

Key exchanges can help here! What you could do is have your service generate a key exchange key pair and provision any new user with the service's public key. This is known as an *authenticated key exchange*; your users know the server's public key, and thus, an active MITM adversary cannot impersonate that side of the key exchange. What a malicious person can do, though, is to perform their own key exchange (as the client side of the connection is not authenticated). By the way, when both sides are authenticated, we call that a *mutually authenticated key exchange*.

This scenario is extremely common, and the key exchange primitive allows it to scale well with an increase of users. But this scenario doesn't scale well if the number of services increase as well! The internet is a good example of this. We have many browsers trying to communicate securely with many websites. Imagine if you had to hardcode the public key of all the websites you might one day visit in your browser and what happens when more websites come online?

While key exchanges are useful, they do not scale well in all scenarios without their sister primitive—the *digital signature*. This is just a teaser though. In chapter 7, you will learn about that new cryptographic primitive and how it helps scaling trust in a system. Key exchanges are rarely used directly in practice, however. They are often just building blocks of a more complex protocol. That being said, they can still be useful on their own in certain situations (for example, as we saw previously against passive adversaries).

Let's now look at how you *would* use a key exchange cryptographic primitive in practice. libsodium is one of the most well known and widely used C/C++ libraries. The following listing shows how you would use libsodium in practice in order to perform a key exchange.

Listing 5.1 A key exchange in C

```
unsigned char client_pk[crypto_kx_PUBLICKEYBYTES];
unsigned char client_sk[crypto_kx_SECRETKEYBYTES];
crypto_kx_keypair(client_pk, client_sk);
```

Generates the client's key pair

```
unsigned char server_pk[crypto_kx_PUBLICKEYBYTES];
obtain(server_pk);
```

We assume that we have some way to obtain the server's public key.

```
unsigned char decrypt_key[crypto_kx_SESSIONKEYBYTES];
unsigned char encrypt_key[crypto_kx_SESSIONKEYBYTES];
```

libsodium derives two symmetric keys instead of one per best practice; each key is used to encrypt a single direction.

```
if (crypto_kx_client_session_keys(decrypt_key, encrypt_key,
    client_pk, client_sk, server_pk) != 0) {
    abort_session();
}
```

We perform a key exchange with our secret key and the server's public key.

If the public key is malformed, the function returns an error.

libsodium hides a lot of details from the developer while also exposing safe-to-use interfaces. In this instance, libsodium makes use of the *X25519 key exchange algorithm*, which you will learn more about later in this chapter. In the rest of this chapter, you will learn about the different standards used for key exchanges, as well as how they work under the hood.

5.2 The Diffie-Hellman (DH) key exchange

In 1976, Whitfield Diffie and Martin E. Hellman wrote their seminal paper on the Diffie-Hellman (DH) key exchange algorithm entitled "New Direction in Cryptography." What a title! DH was the first key exchange algorithm invented and one of the first formalizations of a public key cryptographic algorithm. In this section, I lay out the math foundations of this algorithm, explain how it works, and finally, talk about the standards that specify how to use it in a cryptographic application.

5.2.1 Group theory

The DH key exchange is built on top of a field of mathematics called *group theory*, which is the base of most public key cryptography today. For this reason, I will spend some time in this chapter giving you the basics on group theory. I will do my best to provide good insights on how these algorithms work, but there's no way around it, there is going to be some math.

Let's start with the obvious question: what's a *group*? It's two things:

- A set of elements
- A special binary operation (like + or ×) defined on these elements

If the set and the operation manage to satisfy some properties, then we have a group. And, if we have a group, then we can do magical things . . . (more on that later). Note that DH works in a *multiplicative group*: a group where the multiplication is used as the defined binary operation. Due to this, the rest of the explanations use a multiplicative group as examples. I will also often omit the × symbol (for example, I will write $a \times b$ as ab instead).

I need to be a bit more specific here. For the set and its operation to be a group, they need the following properties. (As usual, I illustrate these properties in a more visual way in figure 5.4 to provide you with more material to grasp this new concept.)

- *Closure*—Operating on two elements results in another element of the same set. For example, for two elements of the group a and b, $a \times b$ results in another group element.

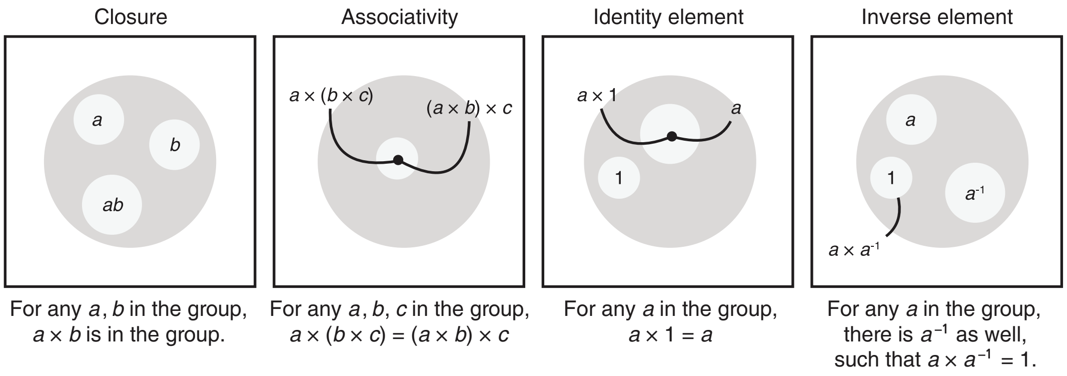

Closure
a
b
ab
For any a, b in the group, a × b is in the group.
Associativity
a × (b × c)
(a × b) × c
For any a, b, c in the group, a × (b × c) = (a × b) × c
Identity element
a × 1
a
1
For any a in the group, a × 1 = a
Inverse element
a
1
a⁻¹
a × a⁻¹
For any a in the group, there is a⁻¹ as well, such that a × a⁻¹ = 1.

Figure 5.4 The four properties of a group: closure, associativity, identity element, and inverse element.

- *Associativity*—Operating on several elements at a time can be done in any order. For example, for three elements of the group a, b, and c, then $a(bc)$ and $(ab)c$ result in the same group element.
- *Identity element*—Operating with this element does not change the result of the other operand. For example, we can define the identity element as 1 in our multiplicative group. For any group element a, we have $a \times 1 = a$.
- *Inverse element*—Existing as an inverse to all group elements. For example, for any group element a, there's an inverse element a^{-1} (also written as $1/a$) such that $a \times a^{-1} = 1$ (also written as $a \times \frac{1}{a} = 1$).

I can imagine that my explanation of a group can be a bit abstract, so let's see what DH uses as a group in practice. First, DH uses a group comprised of the set of strictly positive integers: 1, 2, 3, 4, ···, $p - 1$, where p is a prime number and 1 is the identity element. Different standards specify different numbers for p, but intuitively, it has to be a large prime number for the group to be secure.

Prime numbers

A *prime number* is a number that can only be divided by 1 or by itself. The first prime numbers are 2, 3, 5, 7, 11, and so on. Prime numbers are everywhere in asymmetric cryptography! And, fortunately, we have efficient algorithms to find large ones. To speed things up, most cryptographic libraries will instead look for *pseudo-primes* (numbers that have a high probability of being primes). Interestingly, such optimizations were broken several times in the past; the most infamous occurrence was in 2017, when the ROCA vulnerability uncovered more than a million devices generating incorrect primes for their cryptographic applications.

Second, DH uses the *modular multiplication* as a special operation. Before I can explain what modular multiplication is, I need to explain what *modular arithmetic* is. Modular arithmetic, intuitively, is about numbers that "wrap around" past a certain number called a *modulus*. For example, if we set the modulus to be 5, we say that numbers past 5 go back to 1; for example, 6 becomes 1, 7 becomes 2, and so on. (We also note 5 as 0, but because it is not in our multiplicative group, we don't care too much about it.)

The mathematical way to express modular arithmetic is to take the *remainder* of a number and its *Euclidian division* with the modulus. Let's take, for example, the number 7 and write its Euclidian division with 5 as $7 = 5 \times 1 + 2$. Notice that the remainder is 2. Then we say that $7 = 2 \bmod 5$ (sometimes written as $7 \equiv 2 \pmod 5$). This equation can be read as 7 is congruent to 2 modulo 5. Similarly

- $8 = 1 \bmod 7$
- $54 = 2 \bmod 13$
- $170 = 0 \bmod 17$
- and so on

The classical way of picturing such a concept is with a clock. Figure 5.5 illustrates this concept.

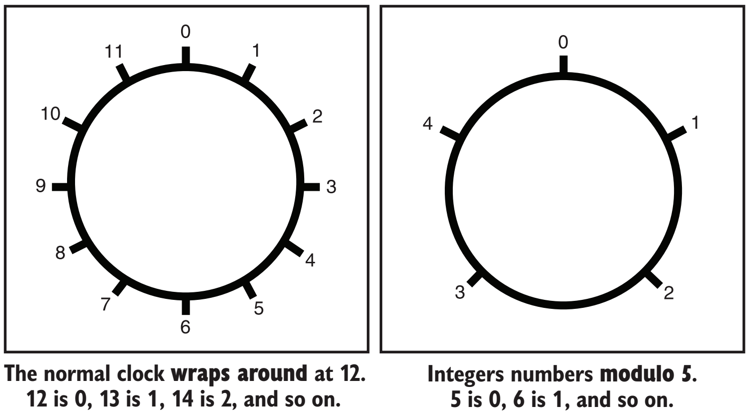

0
1
2
3
4
5
6
7
8
9
10
11
The normal clock wraps around at 12. 12 is 0, 13 is 1, 14 is 2, and so on.
0
1
2
3
4
Integers numbers modulo 5. 5 is 0, 6 is 1, and so on.

Figure 5.5 The group of integers modulo the prime number 5 can be pictured as a clock that resets to 0 after the number 4. Thus 5 is represented as 0, 6 as 1, 7 as 2, 8 as 3, 9 as 4, 10 as 0, and so on.

A modular multiplication is quite natural to define over such a set of numbers. Let's take the following multiplication as an example:

$$3 \times 2 = 6$$

With what you learned previously, you know that 6 is congruent to 1 modulo 5, and thus the equation can be rewritten as:

$$3 \times 2 = 1 \bmod 5$$

Quite straightforward, isn't it? Note that the previous equation tells us that 3 is the inverse of 2 and vice versa. We could also write the following:

$$3^{-1} = 2 \bmod 5$$

When the context is clear, the modulus part (mod 5 here) is often left out from equations. So don't be surprised if I sometimes omit it in this book.

> **NOTE** It happens that when we use the positive numbers modulo a prime number, only the *zero* element lacks an inverse. (Indeed, can you find an element b such that $0 \times b = 1 \bmod 5$?) This is the reason why we do not include zero as one of our elements in the group.

OK, we now have a group, which includes the set of strictly positive integers $1, 2, \cdots, p - 1$ for p a prime number, along with modular multiplication. The group we formed also happens to be two things:

- *Commutative*—The order of operations doesn't matter. For example, given two group elements a and b, then $ab = ba$. A group that has this property is often called a *Galois group*.
- *A finite field*—A Galois group that has more properties, as well as an additional operation (in our example, we can also add numbers together).

Due to the last point, DH defined over this type of group is sometimes called *Finite Field Diffie-Hellman* (FFDH). If you understand what a group is (and make sure you do before reading any further), then a *subgroup* is just a group contained inside your original group. That is, it's a subset of the group elements. Operating on elements of the subgroup results in another subgroup element, and every subgroup element has an inverse in the subgroup, etc.

A *cyclic subgroup* is a subgroup that can be generated from a single *generator* (or *base*). A generator generates a cyclic subgroup by multiplying itself over and over. For example, the generator 4 defines a subgroup consisting of the numbers 1 and 4:

- $4 \bmod 5 = 4$
- $4 \times 4 \bmod 5 = 1$
- $4 \times 4 \times 4 \bmod 5 = 4$ (we start again from the beginning)
- $4 \times 4 \times 4 \times 4 \bmod 5 = 1$
- and so on

> **NOTE** We can also write $4 \times 4 \times 4$ as 4^3.

It happens that when our modulus is prime, every element of our group is a generator of a subgroup. These different subgroups can have different sizes, which we call *orders*. I illustrate this in figure 5.6.

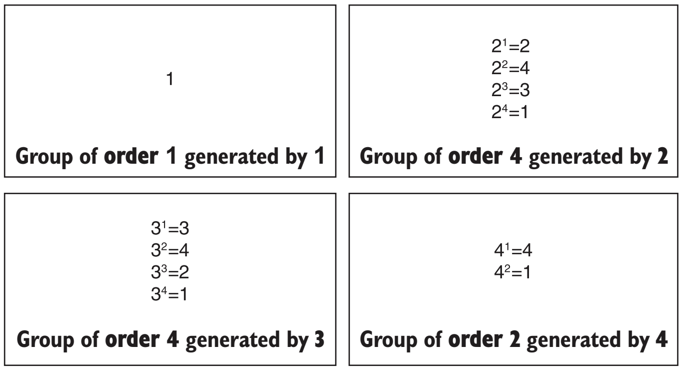

1
Group of order 1 generated by 1
$2^1=2$
$2^2=4$
$2^3=3$
$2^4=1$
Group of order 4 generated by 2
$3^1=3$
$3^2=4$
$3^3=2$
$3^4=1$
Group of order 4 generated by 3
$4^1=4$
$4^2=1$
Group of order 2 generated by 4

Figure 5.6 The different subgroups of the multiplicative group modulo 5. These all include the number 1 (called the *identity element*) and have different orders (number of elements).

All right, you now understand

- A group is a set of numbers with a binary operation that respects some properties (closure, associativity, identity element, inverse element).
- DH works in the Galois group (a group with commutativity), formed by the set of strictly positive numbers up to a prime number (not included) and the modular multiplication.
- In a DH group, every element is a generator of a subgroup.

Groups are the center of a huge amount of different cryptographic primitives. It is important to have good intuitions about group theory if you want to understand how other cryptographic primitives work.

5.2.2 The discrete logarithm problem: The basis of Diffie-Hellman

The security of the DH key exchange relies on the *discrete logarithm problem* in a group, a problem believed to be hard to solve. In this section, I briefly introduce this problem.

Imagine that I take a generator, let's say 3, and give you a random element among the ones it can generate, let's say $2 = 3^x \bmod 5$ for some x unknown to you. Asking you "what is x?" is the same as asking you to find the discrete logarithm of 2 in base 3. Thus, the discrete logarithm problem in our group is about finding out how many times we multiplied the generator with itself in order to produce a given group element. This is an important concept! Take a few minutes to think about it before continuing.

In our example group, you can quickly find that 3 is the answer (indeed, $3^3 = 2 \bmod 5$). But if we picked a much larger prime number than 5, things get much more complicated: it becomes hard to solve. This is the secret sauce behind Diffie-Hellman. You now know enough to understand how to generate a key pair in DH:

1. All the participants must agree on a large prime p and a generator g.
2. Each participant generates a random number x, which becomes their private key.
3. Each participant derives their public key as $g^x \bmod p$.

The fact that the discrete logarithm problem is *hard* means that no one should be able to recover the private key from the public key. I illustrate this in figure 5.7.

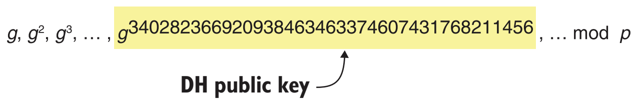

$g, g^2, g^3, \ldots, g^{340282366920938463463374607431768211456}, \ldots \bmod p$
DH public key

Figure 5.7 Choosing a private key in Diffie-Hellman is like choosing an index in a list of numbers produced by a generator *g*. The discrete logarithm problem is to find the index from the number alone.

While we do have algorithms to compute discrete logarithms, they are not efficient in practice. On the other hand, if I give you the solution x to the problem, you have extremely efficient algorithms at your disposal to check that, indeed, I provided you with the right solution: $g^x \bmod p$. If you are interested, the state-of-the-art technique to compute the modular exponentiation is called *square and multiply*. This computes the result efficiently by going through x bit by bit.

> **NOTE** Like everything in cryptography, it is *not impossible* to find a solution by simply trying to guess. Yet, by choosing large enough parameters (here, a large prime number), it is possible to reduce the efficacy of such a search for a solution down to negligible odds. This means that even after hundreds of years of random tries, your odds of finding a solution should still be statistically close to zero.

Great. How do we use all of this math for our DH key exchange algorithm? Imagine that

- Alice has a private key a and a public key $A = g^a \bmod p$.
- Bob has a private key b and a public key $B = g^b \bmod p$.

With the knowledge of Bob's public key, Alice can compute the shared secret as $B^a \bmod p$. Bob can do a similar computation with Alice's public key and his own private key: $A^b \bmod p$. Naturally, we can see that these two calculations end up computing the same number:

$$B^a = (g^b)^a = g^{ab} = (g^a)^b = A^b \bmod p$$

And that's the magic of DH. From an outsider's point of view, just observing the public keys A and B does not help in any way to compute the result of the key exchange $g^{ab} \bmod p$. Next, you will learn about how real-world applications make use of this algorithm and the different standards that exist.

Computational and decisional Diffie-Hellman

By the way, in theoretical cryptography, the idea that observing g^a mod p and g^b mod p does not help you to compute g^{ab} mod p is called the *computational Diffie-Hellman assumption* (CDH). It is often confused with the stronger *decisional Diffie-Hellman assumption* (DDH), which intuitively states that given g^a mod p, g^b mod p, and z mod p, nobody should be able to confidently guess if the latter element is the result of a key exchange between the two public keys (g^{ab} mod p) or just a random element of the group. Both are useful theoretical assumptions that have been used to build many different algorithms in cryptography.

5.2.3 The Diffie-Hellman standards

Now that you have seen how DH works, you can understand that participants need to agree on a set of parameters, specifically on a prime number p and a generator g. In this section, you'll learn about how real-world applications choose these parameters and the different standards that exist.

First things first is the prime number p. As I stated earlier, the bigger, the better. Because DH is based on the discrete logarithm problem, its security is directly correlated to the best attacks known on the problem. Any advances in this area can weaken the algorithm. With time, we managed to obtain a pretty good idea of how fast (or slow) these advances are and how much is enough security. The currently known best practices are to use a prime number of 2,048 bits.

NOTE In general, https://keylength.com summarizes recommendations on parameter lengths for common cryptographic algorithms. The results are taken from authoritative documents produced by research groups or government bodies like the ANSSI (France), the NIST (US), and the BSI (Germany). While they do not always agree, they often converge towards similar orders of magnitude.

In the past, many libraries and software often generated and hardcoded their own parameters. Unfortunately, they were sometimes found to be either weak or, worse, completely broken. In 2016, someone found out that Socat, a popular command-line tool, had modified their default DH group with a broken one a year prior, raising the question whether this had been a mistake or an intentional backdoor. Using standardized DH groups might seem like a better idea, but DH is one of the unfortunate counterexamples. Only a few months after the Socat issue, Antonio Sanso, while reading RFC 5114, found that the standard had specified broken DH groups as well.

Due to all of these issues, newer protocols and libraries have converged towards either deprecating DH in favor of Elliptic Curve Diffie-Hellman (ECDH) or using the groups defined in the better standard, RFC 7919 (https://www.rfc-editor.org/info/rfc7919). For this reason, best practice nowadays is to use RFC 7919, which defines several groups of different sizes and security. For example, ffdhe2048 is the group defined by the 2,048-bit prime modulus:

p = 323170060713110073001535134778251633624880571334 89075174588434139269
80683413621000279205636264016468545855635793533081692882902308057347262
52735547424612457410262025279165729728627063003252634282131457669314142
23654220941111348629991657478268034230553086349050635557712219187890332
72956969612974385624174123623722519734640269185579776797682301462539793
30580152268587307611975324364674758554607150438968449403661304976978128
54295958659597567051283852132784468522925504568272879113720098931873959
14337417583782600027803497319855206060753323412260325468408812003110590
7484281003994966956119696956248629032338072839127039

and with generator $g = 2$

NOTE It is common to choose the number 2 for the generator as computers are quite efficient at multiplying with 2 using a simple left shift (<<) instruction.

The group size (or *order*) is also specified as $q = (p - 1)/2$. This implies that both private keys and public keys will be around 2,048 bits size-wise. In practice, these are quite large sizes for keys (compare that with symmetric keys, for example, that are usually 128-bit long). You will see in the next section that defining a group over the elliptic curves allow us to obtain much smaller keys for the same amount of security.

5.3 The Elliptic Curve Diffie-Hellman (ECDH) key exchange

It turns out that the DH algorithm, which we just discussed, can be implemented in different types of groups, not just the multiplicative groups modulo a prime number. It also turns out that a group can be made from elliptic curves, a type of curves studied in mathematics. The idea was proposed in 1985 by Neal Koblitz and Victor S. Miller, independently, and much later in 2000, it was adopted when cryptographic algorithms based on elliptic curves started seeing standardization.

The world of applied cryptography quickly adopted elliptic curve cryptography as it provided much smaller keys than the previous generation of public key cryptography. Compared to the recommended 2,048-bit parameters in DH, parameters of 256 bits were possible with the elliptic curve variant of the algorithm.

5.3.1 What's an elliptic curve?

Let's now explain how elliptic curves work. First and foremost, it is good to understand that elliptic curves are just curves! Meaning that they are defined by all the coordinates x and y that solves an equation. Specifically, this equation

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

for some a_1, a_2, a_3, a_4, and a_6. Note that for most practical curves today, this equation can be simplified as the *short Weierstrass equation*:

$$y^2 = x^3 + ax + b \text{ (where } 4a^3 + 27b^2 \neq 0\text{)}$$

While the simplification is not possible for two types of curves (called *binary curves* and *curves of characteristic 3*), these are used rarely enough that we will use the Weierstrass form in the rest of this chapter. Figure 5.8 shows an example of an elliptic curve with two points taken at random.

At some point in the history of elliptic curves, it was found that a *group* could be constructed over them. From there, implementing DH on top of these groups was straightforward. I will use this section to explain the intuition behind elliptic curve cryptography.

Groups over elliptic curves are often defined as *additive groups*. Unlike multiplicative groups defined in the previous section, the + sign is used instead.

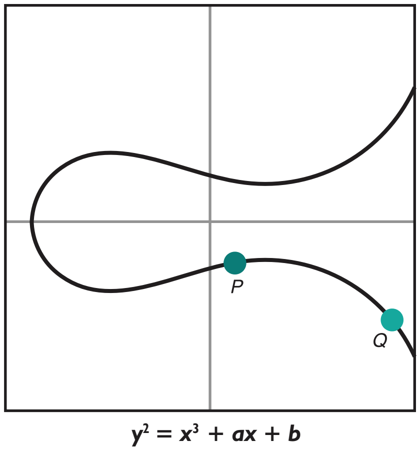


Figure 5.8 One example of an elliptic curve defined by an equation.

NOTE Using an addition or a multiplication does not matter much in practice, it is just a matter of preference. While most of cryptography uses a multiplicative notation, the literature around elliptic curves has gravitated around an additive notation, and thus, this is what I will use when referring to elliptic curve groups in this book.

This time, I will define the operation before defining the elements of the group. Our *addition operation* is defined in the following way. Figure 5.9 illustrates this process.

1 Draw a line going through two points that you want to add. The line hits the curve at another point.

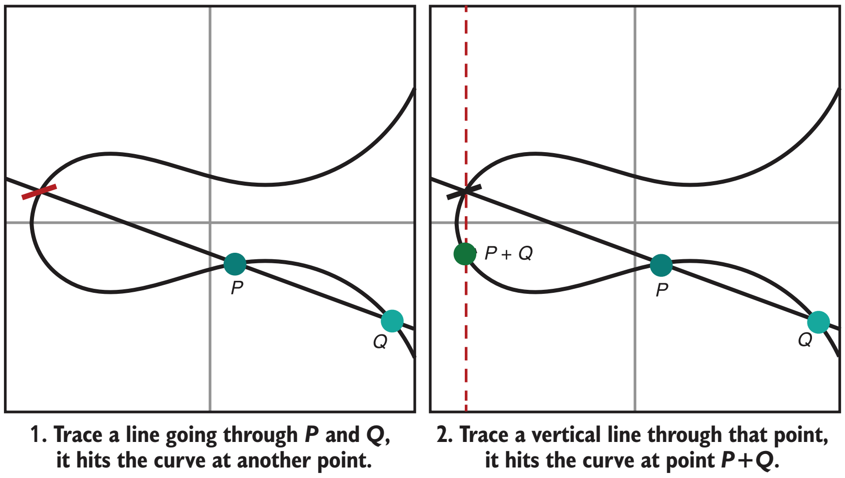
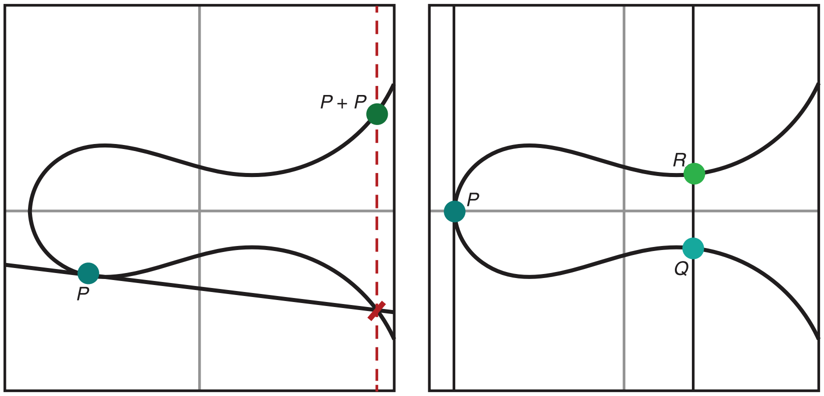

P
Q
1. Trace a line going through *P* and *Q*, it hits the curve at another point.
P + Q
P
Q
2. Trace a vertical line through that point, it hits the curve at point *P*+*Q*.

Figure 5.9 An addition operation can be defined over points of an elliptic curve by using geometry.

2. Draw a vertical line from this newly found point. The vertical line hits the curve in yet another point.
3. This point is the result of adding the original two points together.

There are two special cases where this rule won't work. Let's define these as well:

- *How do we add a point to itself?* The answer is to draw the tangent to that point (instead of drawing a line between two points).
- *What happens if the line we draw in step 1 (or step 2) does not hit the curve at any other point?* Well, this is embarrassing, and we need this special case to work and produce a result. The solution is to define the result as a made-up point (something we make up). This newly invented point is called the *point at infinity* (that we usually write with a big letter O). Figure 5.10 illustrates these special cases.



To add a point to itself, draw the tangent and follow the previous method.

Sometimes adding points results in the point at infinity: *P* + *P* = *O* and *R* + *Q* = *O*.

Figure 5.10 Building on figure 5.9, addition on an elliptic curve is also defined when adding a point to itself or when two points cancel each other to result in the point at infinity (O).

I know this point at infinity is some next-level weirdness, but don't worry too much about it. It is really just something we came up with in order to make the addition operation work. Oh, and by the way, it behaves like a zero, and it is our identity element:

$$O + O = O$$

and for any point P on the curve

$$P + O = P$$

All good. So far, we saw that to create a group on top of an elliptic curve, we need

- An elliptic curve equation that defines a set of valid points.
- A definition of what addition means in this set.
- An imaginary point called a point at infinity.

I know this is a lot of information to unpack, but we are missing one last thing. Elliptic curve cryptography makes use of the previously-discussed type of group defined over a *finite field*. In practice, what this means is that our coordinates are the numbers 1, 2, ⋯, $p - 1$ for some large prime number p. This should sound familiar! For this reason, when thinking of elliptic curve cryptography, you should think of a graph that looks much more like the one on the right in figure 5.11.

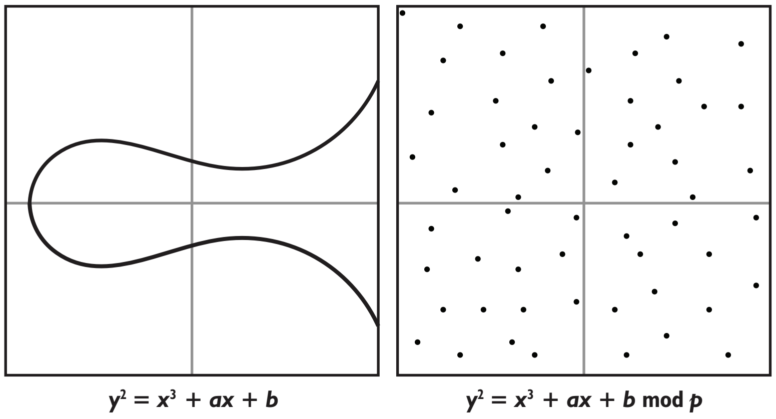


Figure 5.11 Elliptic curve cryptography (ECC), in practice, is mostly specified with elliptic curves in coordinates modulo a large prime number *p*. This means that what we use in cryptography looks much more like the right graph than the left graph.

And that's it! We now have a group we can do cryptography on, the same way we had a group made with the numbers (excluding 0) modulo a prime number and a multiplication operation for Diffie-Hellman. How can we do Diffie-Hellman with this group defined on elliptic curves? Let's see how the *discrete logarithm* works now in this group.

Let's take a point G and add it to itself x times to produce another point P via the addition operation we defined. We can write that as $P = G + \cdots + G$ (x times) or use some mathematical syntactic sugar to write that as $P = [x]G$, which reads x times G. The *elliptic curve discrete logarithm problem* (ECDLP) is to find the number x from knowing just P and G.

NOTE We call $[x]G$ scalar multiplication as x is usually called a scalar in such groups.

5.3.2 How does the Elliptic Curve Diffie-Hellman (ECDH) key exchange work?

Now that we built a group on elliptic curves, we can instantiate the same Diffie-Hellman key exchange algorithm on it. To generate a key pair in ECDH:

1. All the participants agree on an elliptic curve equation, a finite field (most likely a prime number), and a generator G (usually called a *base point* in elliptic curve cryptography).
2. Each participant generates a random number x, which becomes their private key.
3. Each participant derives their public key as $[x]G$.

Because the elliptic curve discrete logarithm problem is hard, you guessed it, no one should be able to recover your private key just by looking at your public key. I illustrate this in figure 5.12.

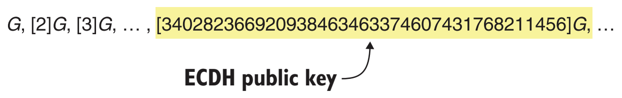

G, [2]G, [3]G, … , [340282366920938463463374607431768211456]G, …
ECDH public key

Figure 5.12 Choosing a private key in ECDH is like choosing an index in a list of numbers produced by a generator (or base point) *G*. The Elliptic Curve Discrete Logarithm Problem (ECDLP) is to find the index from the number alone.

All of this might be a bit confusing as the operation we defined for our DH group was the multiplication, and for an elliptic curve, we now use addition. Again, these distinctions do not matter at all because they are equivalent. You can see a comparison in figure 5.13.

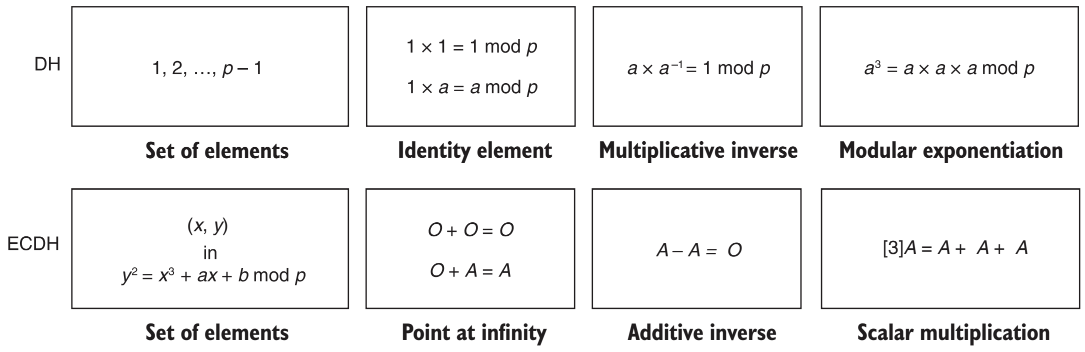


Figure 5.13 Some comparisons between the group used in Diffie-Hellman and the group used in Elliptic Curve Diffie-Hellman (ECDH).

You should now be convinced that the only thing that matters for cryptography is that we have a group defined with its operation, and that the discrete logarithm for this group is hard. For completion, figure 5.14 shows the difference between the discrete logarithm problem in the two types of groups we've seen.

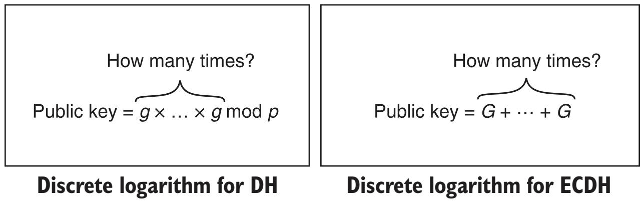

How many times?
Public key = $\underbrace{g \times \ldots \times g}$ mod p
Discrete logarithm for DH
How many times?
Public key = $\underbrace{G + \cdots + G}$
Discrete logarithm for ECDH

Figure 5.14 A comparison between the discrete logarithm problem modulo large primes and the discrete logarithm problem in elliptic curve cryptography (ECC). They both relate to the DH key exchange, as the problem is to find the private key from a public key.

A last note on the theory, the group we formed on top of elliptic curves differs with the group we formed with the strictly positive integers modulo a prime number. Due to some of these differences, the strongest attacks known against DH (known as *index calculus* or *number field sieve* attacks) do not work well on the elliptic curve groups. This is the main reason why parameters for ECDH can be much lower than the parameters for DH at the same level of security.

OK, we are done with the theory. Let's go back to defining ECDH. Imagine that

- Alice has a private key a and a public key $A = [a]G$.
- Bob has a private key b and a public key $B = [b]G$.

With the knowledge of Bob's public key, Alice can compute the shared secret as $[a]B$. Bob can do a similar computation with Alice's public key and his own private key: $[b]A$. Naturally, we can see that these two calculations end up computing the same number:

$$[a]B = [a][b]G = [ab]G = [b][a]G = [b]A$$

No passive adversary should be able to derive the shared point just from observing the public keys. Looks familiar, right? Next, let's talk about standards.

5.3.3 The standards for Elliptic Curve Diffie-Hellman

Elliptic curve cryptography has remained at its full strength since it was first presented in 1985. [. . .] The United States, the UK, Canada and certain other NATO nations have all adopted some form of elliptic curve cryptography for future systems to protect classified information throughout and between their governments.

—NSA ("The Case for Elliptic Curve Cryptography," 2005)

The standardization of ECDH has been pretty chaotic. Many standardization bodies have worked to specify a large number of different curves, which was then followed by many flame wars over which curve was more secure or more efficient. A large amount of research, mostly led by Daniel J. Bernstein, pointed out the fact that a number of curves standardized by NIST could potentially be part of a weaker class of curves only known to NSA.

> *I no longer trust the constants. I believe the NSA has manipulated them through their relationships with industry.*
>
> —Bruce Schneier ("The NSA Is Breaking Most Encryption on the Internet," 2013)

Nowadays, most of the curves in use come from a couple standards, and most applications have fixated on two curves: P-256 and Curve25519. In the rest of this section, I will go over these curves.

NIST FIPS 186-4, "Digital Signature Standard," initially published as a standard for signatures in 2000, contains an appendix specifying 15 curves for use in ECDH. One of these curves, P-256, is the most widely used curve on the internet. The curve was also specified in Standards for Efficient Cryptography (SEC) 2, v2, "Recommended Elliptic Curve Domain Parameters," published in 2010 under a different name, secp256r1. P-256 is defined with the short Weierstrass equation:

$$y^2 = x^3 + ax + b \bmod p$$

where $a = -3$ and

b = 41058363725152142129326129780047268409114441015993725554835256314039
467401291

and

$$p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$$

This defines a curve of prime order:

n = 115792089210356248762697446949407573529996955224135760342422259061068
512044369

meaning that there are exactly n points on the curve (including the point at infinity). The base point is specified as

G = (48439561293906451759052585252797914202762949526041747995844080717082404635286, 36134250956749795798585127919587881956611106672985015071877198253568414405109)

The curve provides 128 bits of security. For applications that are using other cryptographic algorithms providing 256-bit security instead of 128 bits of security (for example, AES with a 256-bit key), P-521 is available in the same standard to match the level of security.

Can we trust P-256?

Interestingly, P-256 and other curves defined in FIPS 186-4 are said to be generated from a *seed*. For P-256, the seed is known to be the byte string

0xc49d360886e704936a6678e1139d26b7819f7e90

I talked about this notion of "nothing-up-my-sleeve" numbers before—constants that aim to prove that there was no room for backdooring the design of the algorithm. Unfortunately, there isn't much explanation behind the P-256 seed other than the fact that it is specified along the curve's parameter.

RFC 7748, "Elliptic Curves for Security," which was published in 2016, specifies two curves: Curve25519 and Curve448. Curve25519 offers approximately 128 bits of security, while Curve448 offers around 224 bits of security for protocols that want to hedge against potential advances in the state of attacks against elliptic curves. I will only talk about Curve25519 here, which is a Montgomery curve defined by the equation:

$$y^2 = x^3 + 486662\ x^2 + x \bmod p, \text{ where } p = 2^{255} - 19$$

Curve25519 has an order

$$n = 2^{252} + 27742317777372353535851937790883648493$$

and the base point used is

$G = (9, 14781619447589544791020593568409986887264606134616475288964881837755586237401)$

The combination of ECDH with Curve25519 is often dubbed *X25519*.

5.4 Small subgroup attacks and other security considerations

Today, *I would advise you to use ECDH over DH* due to the size of the keys, the lack of known strong attacks, the quality of the available implementations, and the fact that elliptic curves are fixed and well standardized (as opposed to DH groups, which are all over the place). The latter point is quite important! Using DH means potentially using broken standards (like RFC 5114 mentioned previously), protocols that are too relaxed (many protocols, like older versions of TLS, don't mandate what DH groups to use), software that uses broken custom DH groups (the socat issue mentioned previously), and so on.

If you do have to use Diffie-Hellman, make sure to *stick to the standards*. The standards I mentioned previously make use of safe primes as modulus: primes of the form $p = 2q + 1$ where q is another prime number. The point is that groups of this form only have two subgroups: a small one of size 2 (generated by –1) and a large one of size q. (This is the best you can get, by the way; there exist no prime-order groups in DH.) The scarcity of small subgroups prevent a type of attack known as *small subgroup attack* (more on that later). Safe primes create secure groups because of two things:

- The order of a multiplicative group modulo a prime p is calculated as p – 1.
- The order of a group's subgroups are the factors of the group's order (this is the Lagrange theorem).

Hence, the order of our multiplicative group modulo a safe prime is $p - 1 = (2q + 1) - 1 = 2q$ that has factors 2 and q, which means that its subgroups can only be of order 2 or q. In such groups, small subgroup attacks are not possible because there are not enough small subgroups. A *small subgroup attack* is an attack on key exchanges in which an attacker sends several invalid public keys to leak bits of your private key gradually, and where the invalid public keys are generators of small subgroups.

For example, an attacker could choose –1 (the generator of a subgroup of size 2) as public key and send it to you. By doing your part of the key exchange, the resulting shared secret is an element of the small subgroup (–1 or 1). This is because you just raised the small subgroup generator (the attacker's public key) to the power of your private key. Depending on what you do with that shared secret, the attacker could guess what it is, and leak some information about your private key.

With our example of malicious public key, if your private key was even, the shared secret would be 1, and if your private key was odd, the shared secret would be –1. As a result, the attacker learned one bit of information: the least significant bit of your private key. Many subgroups of different sizes can lead to more opportunities for the attacker to learn more about your private key until the entire key is recovered. I illustrate this issue in figure 5.15.

While it is always a good idea to verify if the public key you receive is in the correct subgroup, not all implementations do that. In 2016, a group of researchers analyzed 20 different DH implementations and found that none were validating public keys (see "Measuring small subgroup attacks against Diffie-Hellman" from Valenta et al.) Make sure that the DH implementations you're using do! You can do this by raising the public key to the order of the subgroup, which should give you back the identity element if it is an element of that subgroup.

On the other hand, elliptic curves allow for groups of prime order. That is, they have no small subgroups (besides the subgroup of size 1 generated by the identity element), and thus, they are secure against small subgroup attacks. Well, not so fast . . . In 2000, Biehl, Meyer, and Muller found that small subgroup attacks are possible even in such prime-order elliptic curve groups due to an attack called *invalid curve attack*.

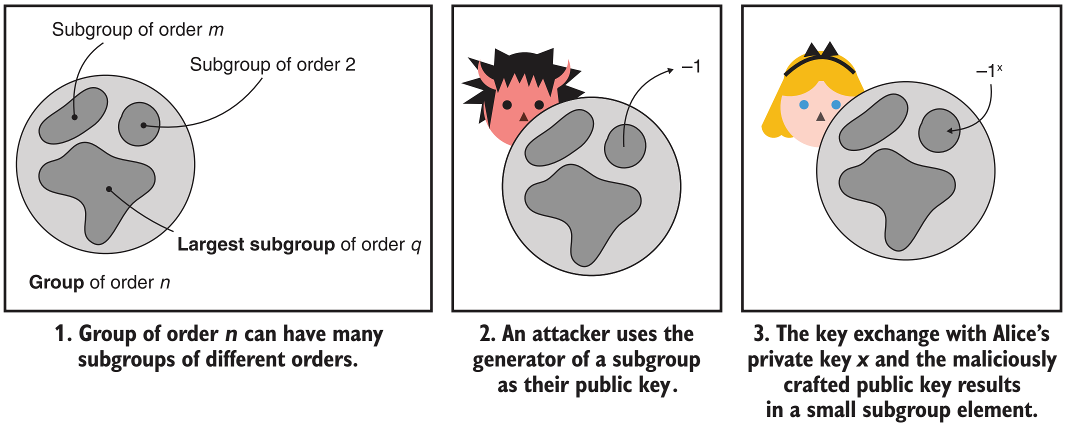

Subgroup of order m
Subgroup of order 2
Largest subgroup of order q
Group of order n
-1
-1^x
1. Group of order n can have many subgroups of different orders.
2. An attacker uses the generator of a subgroup as their public key.
3. The key exchange with Alice's private key x and the maliciously crafted public key results in a small subgroup element.

Figure 5.15 A small subgroup attack impacts DH groups that have many subgroups. By choosing generators of small subgroups as public keys, an attacker can leak bits of someone's private key little by little.

The idea behind invalid curve attacks is the following. First, the formulas to implement scalar multiplication for elliptic curves that use the short Weierstrass equation $y^2 = x^3 + ax + b$ (like NIST's P-256) are independent of the variable b. This means that an attacker can find different curves with the same equation except for the value b, and some of these curves will have many small subgroups. You probably know where this is going: the attacker chooses a point in another curve that exhibits small subgroups and sends it to a targeted server. The server goes on with the key exchange by performing a scalar multiplication with the given point, effectively doing a key exchange on a different curve. This trick ends up re-enabling the small subgroup attack, even on prime-order curves.

The obvious way to fix this is to, again, validate public keys. This can be done easily by checking if the public key is not the point at infinity and by plugging the received coordinates into the curve equation to see if it describes a point on the defined curve. Unfortunately, in 2015, Jager, Schwenk, and Somorovsky showed in "Practical Invalid Curve Attacks on TLS-ECDH" that several popular implementations did not perform these checks. If using ECDH, I would advise you to use the X25519 key exchange due to the quality of the design (which takes into account invalid curve attacks), the quality of available implementations, and the resistance against timing attacks by design.

Curve25519 has one caveat though—it is not a prime-order group. The curve has two subgroups: a small subgroup of size 8 and a large subgroup used for ECDH. On top of that, the original design did not prescribe validating received points, and libraries, in turn, did not implement these checks. This led to issues being found in different types of protocols that were making use of the primitive in more exotic ways. (One of these I found in the Matrix messaging protocol, which I talk about in chapter 11.)

Not verifying public keys can have unexpected behaviors with X25519. The reason is that the key exchange algorithm does not have *contributory behavior*: it does not allow both parties to contribute to the final result of the key exchange. Specifically, one of the participants can force the outcome of the key exchange to be all zeros by sending a point in the small subgroup as public key. RFC 7748 does mention this issue and proposes to check that the resulting shared secret is not the all zero output, yet lets the implementer decide to do the check or not! I would recommend making sure that your implementation performs the check, although it's unlikely that you're going to run into any issues unless you use X25519 in a nonstandard way.

Because many protocols rely on Curve25519, this has been an issue for more than just key exchanges. *Ristretto*, the internet draft soon-to-be RFC, is a construction that adds an extra layer of encoding to Curve25519, effectively simulating a curve of prime order (see https://tools.ietf.org/html/draft-hdevalence-cfrg-ristretto-01). The construction has been gaining traction as it simplifies the security assumptions made by other types of cryptographic primitives that want to benefit from Curve25519 but want a prime-order field.

Summary

- Unauthenticated key exchanges allow two parties to agree on a shared secret, while preventing any passive man-in-the-middle (MITM) attacker from being able to derive it as well.
- An authenticated key exchange prevents an active MITM from impersonating one side of the connection, while a mutually authenticated key exchange prevents an active MITM from impersonating both sides.
- One can perform an authenticated key exchange by knowing the other party's public key, but this doesn't always scale and signatures will unlock more complex scenarios (see chapter 7).
- Diffie-Hellman (DH) is the first key exchange algorithm invented and is still widely used.
- The recommended standard to use for DH is RFC 7919, which includes several parameters to choose from. The smallest option is the recommended 2,048-bit prime parameter.
- Elliptic Curve Diffie-Hellman (ECDH) has much smaller key sizes than DH. For 128 bits of security, DH needs 2,048-bit parameters, whereas ECDH needs 256-bit parameters.
- The most widely used curves for ECDH are P-256 and Curve25519. Both provide 128 bits of security. For 256-bit security, P-521 and Curve448 are available in the same standards.
- Make sure that implementations verify the validity of public keys you receive as invalid keys are the source of many bugs.